Resonance Algorithm: A New Look at the Shortest Path Problem

Yu LIU[1,*], **Qiguang LIN**[2], **Binbin HONG**[3], **Daniel HJERPE**[4] **& Xiaofeng LIU**[2,5,*]

[1] International Academic Center of Complex Systems, Beijing Normal University at Zhuhai, Zhuhai 519087, Guangdong, China.
[2] College of IoT Engineering, Hohai University, Changzhou 213022, Jiangsu, China.
[3] Institute of Microscale Optoelectronics, Shenzhen University, Shenzhen 518060, Guangdong, China.
[4] Ericsson AB, Kista 16483, Sweden.
[5] Jiangsu Key laboratory of Special Robotic Technologies, Changzhou 213022, Jiangsu, China.

Author Contribution Statement:
Yu LIU: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing.
Qiguang LIN: Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – review & editing.
Binbin HONG: Investigation, Methodology, Writing – review & editing.
Daniel HJERPE: Data curation, Formal analysis, Investigation, Visualization.
Xiaofeng LIU: Investigation, Project administration, Resources, Supervision, Writing – review & editing.

Abstract

The shortest path problem (SPP) is a classic problem and appears in a wide range of applications. Although a variety of algorithms already exist, new advances are still being made, mainly tuned for particular scenarios to have better performances. As a result, they become more and more technically complex and sophisticated. Here we developed a novel nature-inspired algorithm to compute all possible shortest paths between two nodes in a graph: *Resonance Algorithm* (RA), which is surprisingly simple and intuitive. Besides its simplicity, RA turns out to be much more time-efficient for large-scale graphs than the extended Dijkstra's algorithm (such that it gives all possible shortest paths). Moreover, RA can handle any undirected, directed, or mixed graphs, irrespective of loops, unweighted or positively-weighted edges, and can be implemented in a fully decentralized manner. These good properties ensure RA a wide range of applications.

Keywords: Dijkstra's algorithm; Large-scale graph; Nature-inspired algorithm; Decentralized algorithm; Fermat's principle

Abbreviations: shortest path problem (SPP); resonance algorithm (RA).

1. Introduction

The shortest path problem (SPP), i.e., to find a path between two nodes in a graph such that the length (or weights) of the path is minimized, is a classic problem in graph theory and computer

* **Corresponding Authors:** yu.ernest.liu@bnu.edu.cn (Yu LIU); xfliu@hhu.edu.cn (Xiaofeng LIU)
The animation and the codes can be found at www.wuyichen.org/resalg or in the supplementary materials.

science [1], with a wide range of applications such as route planning [2–4], network routing [5–7] and task planning [8,9]. There is a variety of algorithms to solve SPP, including the classic Dijkstra's algorithm [10] (which is commonly used and lays the basis for many other methods [11–13]), Bellman-Ford algorithm [14], Chan's fast algorithm (in $O(n^3/\log n)$ time) [15], etc.

Although SPP has been studied extensively, new advances are still being made, to have better performances in specific scenarios, including rapid explorations [16,17], solving multi-objective problems [18,19], handling dynamic networks [20–22] and stochastic situations [23,24], etc. More specifically, for example, Noto and Sato proposed an algorithm to compute paths as close as possible to the optimal solution, but in a much shorter time, to handle real-time problems [17]. Galán-García *et al.* developed Probabilistic Extension of Dijkstra's Algorithm to calculate not only the shortest path but also the second-, third- or fourth-shortest path, by taking into account traffic flows, which is extremely helpful in realistic car navigation [25]. Moreover, by taking off the "Fibonacci heap" in the original Dijkstra's algorithm, Xu *et al.* substantially improved the efficiency of Dijkstra's algorithm for sparse networks, to which the road traffic network belongs [26].

While finding one shortest path has a wide range of applications (as examples above), finding all possible shortest paths (without missing any one) is also a significant question to ask in various scenarios [27], e.g., when searching a path that should satisfy other conditions beyond having the minimal length, e.g., in the power line routing problem [28]. For another example, in the biological sequence alignment problem (which can be reduced to SPP), we need to investigate alternative optimal solutions in order to check the sensitivity of this problem [29]. Moreover, researchers developed a computational approach to identify liver cancer related genes, where the essential step is to find all (not just only one) shortest paths in the protein-protein interaction network [30].

The most straightforward approach to find all shortest paths is to simply use breadth-first search, which is inefficient, though [31]. Alternatively, Dijkstra's algorithm can be extended to return all possible shortest paths [32] (this extended algorithm will be compared with our proposed algorithm later in this paper). Last but not least, we can "hack" the algorithm that solves the *k shortest paths problem* [33] (i.e., to list k paths in descending lengths where k is a predefined positive integer parameter) so that k is not predefined and the algorithm continues to iterate until the newly-returned path is strictly longer than the one returned in the last iteration.

In this paper, we propose a novel and intuitive, but also surprisingly simple algorithm that finds all possible shortest paths between two nodes in a graph. We call this algorithm *Resonance Algorithm* (RA). First of all, we recommend the reader to check out the 50-second animation of RA at www.wuyichen.org/resalg or in the supplementary materials, for a general idea. Generally speaking, RA can be summarized in three points:

- From the origin, each node sends signals to all of its neighbors when it receives one for the first time, until the destination receives the signal, which we call the *forward process*;
- Likewise, we have the *backward process* but from the destination to the origin;
- All possible shortest paths are the intersections of the forward process and the time-reversed backward process.

Intriguingly, RA might be considered as a metaphor of the combination of Fermat's principle [34] (i.e., the path taken by a ray is always the one that takes the least time) and the probability amplitude interpretation of the wave function in quantum mechanics [35], which we will discuss in Appendix A.

Despite the potential links with natural phenomena, RA turns out to have merits in various aspects, as well. First of all, RA can deal with any undirected, directed or mixed graphs, with or without loops, with unweighted or positively weighted edges. Moreover, as the number of nodes in the graph increases, RA out-competes the extended Dijkstra's algorithm (such that all possible shortest paths can be found, rather than just one shortest path as the classic Dijkstra) in time efficiency. Last but not least, RA does not require the information of the whole graph to be stored in one central agent, and it can thus be implemented in a decentralized manner, that is, the

nodes collectively determine the shortest paths. This is very useful in scenarios where the central agent cannot keep track of the whole graph or do not simply exist, or where nodes are constantly joining and leaving the graph.

This paper is organized as follows. In Section 2, we go through a typical example to illustrate the general scheme of RA. In Section 3, we explain in details about how to implement RA by matrix, one of many possible implementations. Next, in Section 4, we compare RA with Dijkstra's algorithm intuitively first, and then statistically in time efficiency where we shall see that RA wins. In the end, we briefly discuss how to implement RA in a decentralized manner, and then draw the conclusion.

2. Resonance algorithm (RA): general scheme

To illustrate RA, take the undirected and weighted graph shown in Fig. 1(a) as an example (the weighting is represented by the length of edges). Now, the following three subsections elaborate RA step by step, also referring to the animation mentioned above (at www.wuyichen.org/resalg or in the supplementary materials).

(a) Forward process: from origin to destination

- To begin with ($t=0$), the origin node A sends signals to all of its neighbors simultaneously, namely nodes C and S.
- At $t=1$, nodes C and S receive the signal. Immediately, C sends signals to its neighbors D and E; and S sends signals to its neighbors R and Q. Note that they do not need to send signals to the node where the signal came from.
- At $t=2$, D, E, R and Q receive the signal. Immediately, D sends a signal to E; E sends signals to D and F; R sends a signal to H; and Q sends signals to N and P.
- At $t=3$, E, D, F, H, N and P receive the signal. Note that, although D and E receive signals again, they do not need to send signals because this is not the first time they receive them (which means D and E will never send signals again). Nodes F, H, N, and P send signals to their neighbors, immediately.
- This process continues, until the destination node B receives a signal. This is then the *forward process*, denoted as $\mathcal{X}$.

(b) Backward process: from destination to origin

In the *backward process*, the signal is sent from the destination node B to the origin node A, but all of the nodes obey the exact rules as in the forward process. We denote the backward process as $\mathcal{Y}$.

(c) Intersection of forward and backward processes

This is the last step of RA, after which all possible shortest paths will reveal themselves:

- In this step, we simultaneously play the animation of the forward process $\mathcal{X}$ and the backward process $\mathcal{Y}$. But note that the key is to play $\mathcal{X}$ normally but play $\mathcal{Y}$ reversely, namely, in a time-reversed manner.
- At each frame during playing, we mark the signals that appear at the same position in both processes.
- In the end, the paths covered by all of these marked signals are the shortest paths (referring to the highlighted yellow paths in the animation, also shown in Fig. 1(b)).

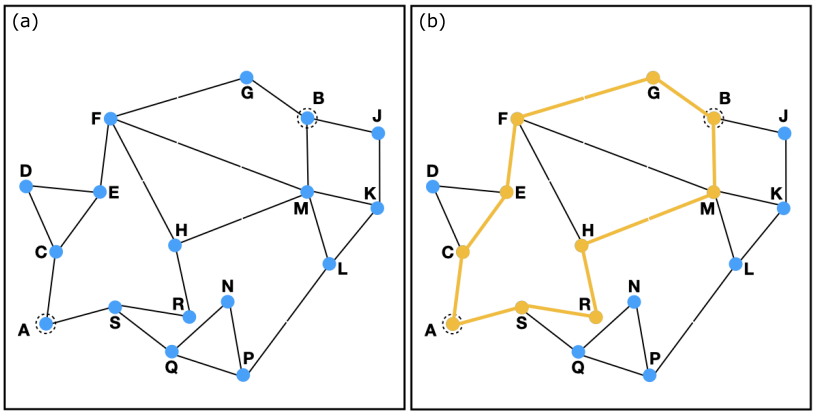

Figure 1. The exemplified graph to illustrate RA, also referring to the animation at www.wuyichen.org/resalg or in the supplementary materials. (a) The question is to find all possible shortest paths from the origin node A to the destination node B. This graph is undirected and weighted, represented by lengths of the edges, which can be interpreted as the time needed to move from one end to the other, e.g., it takes 1 unit of time (e.g., second) to move from A to C, 1 second from R to H, 2 seconds from M to H and 3 seconds from F to M. Here the weights are assumed to be integers but they could be any positive values. (b) The highlighted yellow paths are the two and the only two shortest paths from node A to B.

3. Resonance algorithm (RA): Implemented by matrix

In this section, we explain in details about how to implement RA by matrix, one of many possible implementations, also referring to the MATLAB code at www.wuyichen.org/resalg.

(a) Forward process: from origin to destination (matrix)

First of all, we initialize a zero matrix (denoted as $\mathbf{X}$ and each entry is denoted as $X_{i,j}$) where one row stands for one node and one column stands for one time point. $\mathbf{X}$ records the time point when a node sends signals, where "1" represents signals sent and "0" otherwise.

At $t=0$, we set $X_{A,0}=1$, meaning that the origin node A sends signals to all of its neighbors. Its neighbors C and S will receive this signal at $t=1$ (as the weights of both edges are 1). We then set $X_{C,1}=X_{S,1}=1$, as shown in Fig. 2(a) ("0" is always omitted to write).

Now, repeat the following two procedures until the destination node B receives signals: (i) Set the time to $t+1$, and check the whole column of $(t+1)$ to see which entries are 1, meaning that they receive signals at $(t+1)$ and will send signals to all of their neighbors immediately (denote each neighbor as v); (ii) For each v, there is a specific time point (denoted t_v) it receives the signal, so we mark each $X_{v,t_v}=1$. But note that (1) if $X_{v,\tau}=1$ for any $\tau<t_v$ (meaning that v would send signals before t_v), do not mark it; and (2) if v will receive signals at several time points after $(t+1)$, only mark the earliest time point and change other entries to 0. Specifically:

- At $t=1$, we have $X_{C,1}=X_{S,1}=1$ which means that C and S receive signals at $t=1$ (Fig. 2(b)) and they will send signals to their neighbors immediately. For node C, its neighbors D and E will receive signals at $t=2$, as the weights of both edges are 1, so we mark $X_{D,2}=X_{E,2}=1$ (although node A is also C's neighbor, it has already sent signals, so leave it). Similarly, for node S, we mark its neighbors $X_{Q,2}=X_{R,2}=1$.
- At $t=2$, we see that D, E, R and Q receive the signals (Fig. 2(c)), and will send signals to their neighbors immediately. So, we mark their corresponding neighbors at the time point they receive signals, namely, $X_{F,3}=X_{H,3}=X_{N,3}=X_{P,3}=1$.
- At $t=3$, F, H, N and P receive the signals (Fig. 2(d)). Likewise, we mark their corresponding neighbors at the corresponding time point $X_{G,5}=X_{L,5}=X_{M,5}=1$

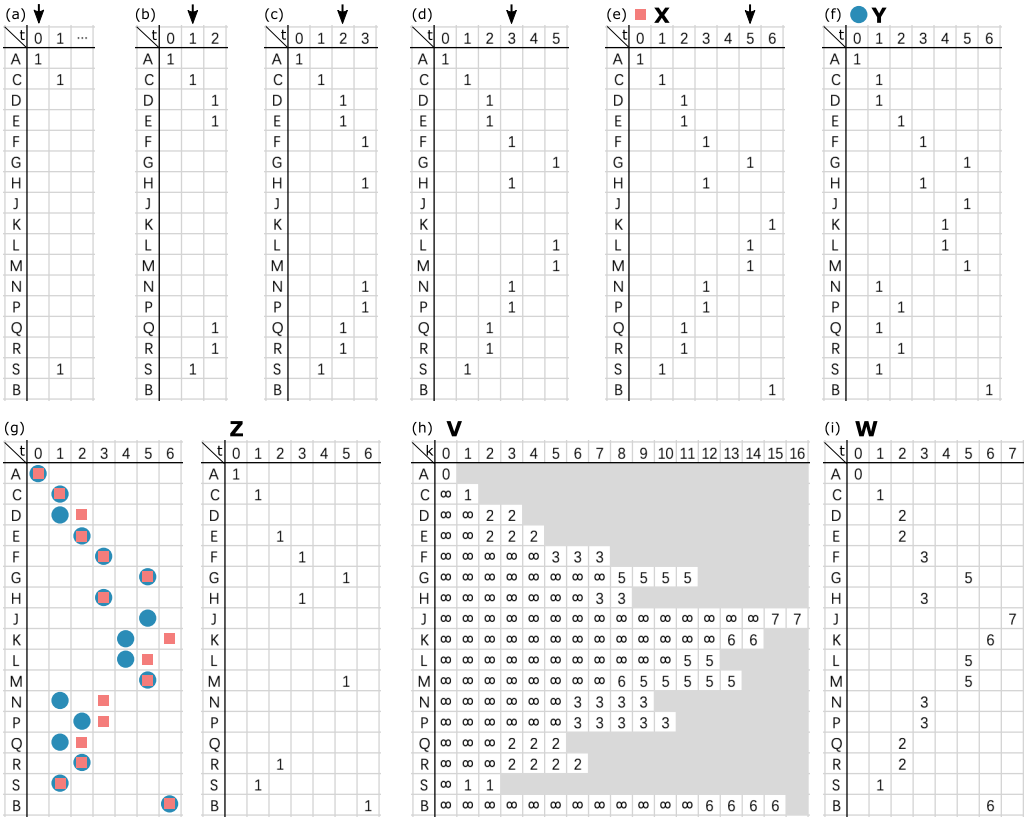

Figure 2. The matrix implementation of RA ("0" is omitted in these matrices). (a)-(e) illustrate how to obtain the matrix $\mathbf{X}$ for the forward process. (f) shows the matrix $\mathbf{Y}$ for the backward process. (g) illustrates the intersection operation where the red squares represent the entries where $X_{i,j} = 1$, and the blue circles represent the entries where $Y_{i,j} = 1$. The entries with both a red square and a blue circle will be set to 1. Then we obtain the matrix $\mathbf{Z}$. (h) The matrix implementation of Dijkstra's algorithm for the same graph (Fig. 1(a)). The entry records the tentative distance from the origin to this node, and the ones marked in gray indicate that they are visited. (i) To compare with RA, we rewrite the matrix $\mathbf{V}$ in terms of time points t instead of operation steps k, into the matrix $\mathbf{W}$.

(notice the weighted edges, i.e., it takes 2 units of time from F to H and G, from H to F and M, from P to L, and it takes 3 units of time from F to M). Note that the signal that F sent arrives at M at $t = 6$ while the signal that H sent arrives at M at $t = 5$, but we will only mark the one that arrives first, i.e., only set $X_{M,5} = 1$.

- At $t = 4$, no node receives signals, so we directly move to the next time point.
- At $t = 5$, G, L and M receive signals for the first time (Fig. 2(e)). Likewise, we mark their corresponding neighbors $X_{B,6} = X_{K,6} = 1$.
- At $t = 6$, the destination node B receives the signal, so the forward process stops. We then obtained the matrix $\mathbf{X}$ for the forward process, as shown in Fig. 2(e).

(b) Backward process: from destination to origin (matrix)

In the backward process, the rules each node follows are exactly the same as that in the forward process, except that (1) the signal is sent from the destination B to the origin A, and that (2) the time goes backwards, i.e., t starts from 6 (the time point when node B receives the signal in the forward process), to 5, 4, ..., till 0. It is automatically guaranteed that at $t = 0$, the origin A will

receive the signal. In the end, we can obtain the matrix $\mathbf{Y}$ for the backward process, as shown in Fig. 2(f).

(c) Intersection of forward and backward processes (matrix)

The last step is to compute the intersection of $\mathbf{X}$ and $\mathbf{Y}$: Create a matrix $\mathbf{Z}$ and set each entry of $\mathbf{Z}$ to be equal to the logical conjunction of the corresponding entries of $\mathbf{X}$ and $\mathbf{Y}$. That is, $Z_{i,j} = X_{i,j} \wedge Y_{i,j}$, where $\wedge$ represents the logical conjunction operator, i.e., $0 \wedge 0 = 1 \wedge 1 = 1$ while $0 \wedge 1 = 1 \wedge 0 = 0$. The matrix $\mathbf{Z}$ is all we need, that contains the information of all possible shortest paths (Fig. 2(g)). From $\mathbf{Z}$, we can visualize the paths as shown in Fig. 1(b) or present them in other ways.

4. Comparison with Dijkstra's algorithm

Dijkstra's algorithm is the most classic and commonly used algorithm for SPP [10,11,36]. One difference between Dijkstra's algorithm and RA is that the latter returns all possible shortest paths while the former returns one shortest path (for the classic version, yet extensions can be made to return all possible shortest paths [32]). In this section, we will first take the classic version of Dijkstra's algorithm, and compare it with RA for similarities and differences in the searching process. And then, we will extend the classic Dijkstra's algorithm to return all possible shortest paths, and compare its running time with RA.

(a) Similarities and differences in the searching process

In general, RA and Dijkstra's algorithm have a few points in common: (1) If a node is currently in focus, the next step is to check all of its neighbors: the former does it by sending signals to all of its neighbors, while the latter does it by updating the distance from the origin to the neighbor. (2) They both ignore certain nodes in future searching processes: RA ignores the nodes that have sent signals, while Dijkstra's algorithm ignores the nodes that have been visited. (3) RA stops searching as long as the destination receives signals, while Dijkstra's algorithm stops when all nodes have been visited.

To compare the two algorithms, we now apply Dijkstra's algorithm on the exemplified graph in Fig. 1(a), and use the similar matrix notation as in Section 3. First of all, we initialize a matrix $\mathbf{V}$ where one row stands for one node, and one column stands for one operation step, denoted k (referring to the matrix shown in Fig. 2(h), which evolves as the algorithm proceeds, but we currently only look at column $k = 0$). Each entry of the matrix records the *tentative distance* from the origin to this node. For column $k = 0$ (initialization), we set the entry for the origin A to 0, and other entries to infinity ∞. Mark all nodes as *unvisited*, whereas *visited* nodes are marked in grey as we shall see later.

- We are first at the initial operation step (column $k = 0$). Select one unvisited node that is marked with the smallest tentative distance, node A in this case. Consider all of its unvisited neighbors, nodes C and S in this case, and calculate their tentative distances, both of which are $0 + 1 = 1$ where 0 is the current tentative distance for A, and 1 is the weight of the edge from A to C or S. Since the new tentative distance (namely 1) is smaller than the current value ∞, we update the corresponding entries of the next operation step to 1, i.e., $V_{C,1} = V_{S,1} = 1$. Then, at column $k = 1$, we mark node A visited (denoted by grey color, and note that A is visited since then) and keep other entries unchanged. Finally, matrix $\mathbf{V}$'s column $k = 1$ is updated.
- Now, we are at column $k = 1$. Select one unvisited node with the smallest tentative distance. Either C or S works, and we can choose C first. Consider all of C's unvisited neighbors (namely, D and E), and calculate their tentative distances, both being $1 + 1 = 2$.

Since 2 is smaller than the current value ∞, update the entries $V_{D,2} = V_{E,2} = 2$ (and other entries unchanged). Finally, mark node C visited.

- Then, we are at column $k = 2$. Select one unvisited node with the smallest tentative distance, S in this case. Calculate the tentative distances of all of S's unvisited neighbors (namely, Q and R), both being $1 + 1 = 2 < \infty$. Then update $V_{Q,3} = V_{R,3} = 2$, and mark S visited.
- This process continues, until all node are marked visited. As we see in Fig. 2(h), it finishes at $k = 16$ in this case.

In order to compare $\mathbf{V}$ with the matrix obtained from RA, we rewrite $\mathbf{V}$ in terms of time points t instead of operation steps k, into the matrix $\mathbf{W}$ (Fig. 2(i)), that is, for each node, if the rightmost entry is m in $\mathbf{V}$, we write m in $\mathbf{W}$ at the mth column and the corresponding row.

We can see that $\mathbf{W}$'s non-empty entries and $\mathbf{X}$'s value-1 entries have exactly the same positions ($\mathbf{W}$ has an extra 7th column due to the fact that all nodes have to be visited in Dijkstra's algorithm). That means, the forward process of RA (as $\mathbf{X}$ is the resulted matrix) and Dijkstra's algorithm are similar, essentially.

The difference between the two algorithms comes after we obtain this matrix. To work out the shortest paths, RA computes the intersection of the forward and the backward process, while Dijkstra's algorithm recursively traces the path of all nodes.

Due to the similar process mentioned above, the two algorithms would have similar algorithmic complexity. On one hand, RA does not require to either record and update each node's tentative distance or memorize the paths visited, which may reduce the running time; while on the other hand, it requires an extra backward process which may double the running time (although there is no effect on the time complexity). So, in order to have a better quantitative comparison, we will investigate their running times systematically, in the next subsection.

(b) Running time statistics

In order to have a fair comparison, we first extend Dijkstra's algorithm of the classic version (which only gives one shortest path) so that it gives all possible shortest paths [32] (see Appendix B for details), just as what RA does. For both algorithms, we wrote the codes in MATLAB, with sufficient optimization (see the codes at www.wuyichen.org/resalg). Now we run the codes on different random graphs (undirected and weighted) with the number of nodes varying.

First of all, we have checked that both algorithms give identical answers all the time. Then, the running time statistics is shown in Fig. 3. We can see that the extended Dijkstra's algorithm is fast when the graph is small, but the running time grows fast as the number of nodes /edges increases; while the running time of RA grows much slower. Eventually, the temporal performance of RA out-competes the extended Dijkstra's algorithm (although the original motivation for RA is not the speed *per se*). That means RA would be very useful in scenarios where the number of nodes is enormous and all possible shortest paths are demanded.

5. Conclusion

We developed a novel, intuitive and surprisingly simple algorithm, *Resonance Algorithm* (RA), to compute all of the possible shortest paths between two nodes in a graph, which is inspired by Fermat's principle (i.e., the path taken by a ray is always the one that takes the least time) and the probability amplitude interpretation of the wave function in quantum mechanics, as the wave function always experiences every possible path (see Appendix A for discussions).

The basic idea of RA is that if each node sends signals to all of its neighbors immediately after it receives one, then when the destination receives the signal for the first time, the signal must have traveled through the shortest path to reach the destination. The information about the shortest paths has already been stored in this process, yet the problem is how to extract it. On the other hand, if the signal is sent from the destination to the origin, there must be signals traveling

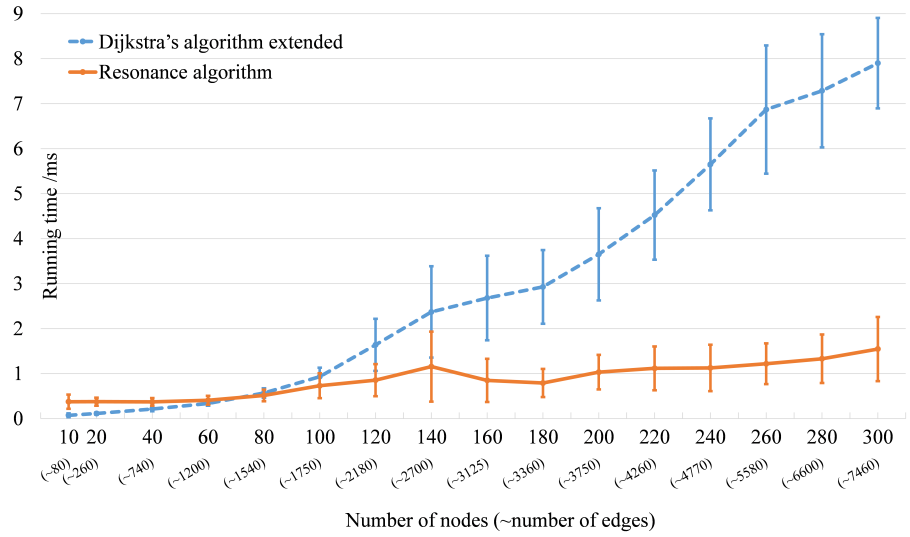

Figure 3. Running time statistics of RA and the extended Dijkstra's algorithm. The curve is the mean running time while the error bar represents the standard error. For a fixed number of nodes (x-axis value), we used 50 randomly generated graphs, all of which are sparse graphs (but all nodes are connected), i.e., the number of edges is much smaller than the number of nodes squared, of which the graph in Fig. 1(a) is a typical example. The MATLAB code was run on a computer with the Intel(R) Core(TM) i5-5200U CPU of 2.20GHz, with 4GB RAM (the running time might be different on different computers but the trend is the same).

through the shortest path, too. So, the intersection of the forward process and the time-reversed backward process may reveal the shortest paths.

RA can handle any undirected, directed, or mixed graphs, with or without loops, with unweighted or positively weighted edges. In addition, it can be implemented in a fully decentralized way, because each node acts as an independent agent, obeys identical rules, and its behavior depends only on the local information, i.e., which node it is linked to and when it receives signals. This would be very useful in scenarios where the central agent does not exist or cannot keep track of the whole graph, or where nodes are constantly joining and leaving the graph. Here, we provide a recipe for the decentralized version: (1) Each node only records who it is linked to, namely all of its neighbors (so there is no need for the whole graph to be hold by a central agent); (2) The signal any node sends should contain the information about where and when it was sent from; (3) After the destination receives the signal in the forward process and the origin receives the signal in the backward process, the shortest paths can be readily worked out from the information contained in the two final signals.

Not only does RA provide a new look at SPP, but practically speaking, it is also found to be more time-efficient than the extended Dijkstra's algorithm (such that it gives all possible shortest paths instead of just one) when the graph is large. Its advantage gets larger and larger as the graph expands. Thus, RA would have a wide range of applications in scenarios where the graph is huge and all possible shortest paths are demanded.

Acknowledgments

This work was supported in part by the National Key Research & Development Program 2018AAA0100803, the Key Research and Development Program of Jiangsu under grant BK20192004, BE2018004-04, and the Guangdong Forestry Science and Technology Innovation Project under Grant 2020-KJCX005, the Projects of International Cooperation and Exchanges of

Changzhou under grant CZ20200035, the National Natural Science Foundation of China (Grant No. 61803381).

Competing interests

The authors declare that they have no competing interests.

Appendix

A. Resonance algorithm's potential links to Fermat's principle and quantum mechanics

In RA, the process that each node sends signals to all of its neighbors when it receives one can be considered as that signals are experiencing every possible path. On the other hand, as long as the origin and the destination are determined, the shortest path (equivalently, the path that takes the least time) is automatically determined. Intriguingly, it seems like a metaphor of Fermat's principle [34] which says that the path taken by a ray (or other types of waves in general) is always the one that takes the least time.

Let us consider the gravity between Earth and the Moon. A naive way to think about it is that a gravitational field is first created by Earth; the Moon perceives this field around itself, and this surrounding gravitational field determines the Moon's movement. But, inspired by RA, we can also consider Earth and the Moon simultaneously. That is, they both create their own gravitational field that expands outwards in all directions at the speed of light; and when either field reaches the other object, the interaction forms. This mechanism guarantees that the interaction forms in the quickest way (indeed, natural interactions occur in the quickest way) since the gravitational fields are experiencing every possible path before the interaction forms, just as RA.

Now, from another perspective, let us consider the wave function of a free particle X in quantum mechanics [35], i.e., $\psi(\boldsymbol{r},t) = Ae^{i(\boldsymbol{k}\cdot\boldsymbol{r}-\omega t)}$ where $\boldsymbol{r}$ is X's position, t is the time, A is the amplitude of the wave function, $\boldsymbol{k}$ is the wave vector, namely the direction X travels, and ω is the angular frequency. We can see that the complex conjugate of $\psi(\boldsymbol{r},t)$ is $\psi^*(\boldsymbol{r},t) = Ae^{-i(\boldsymbol{k}\cdot\boldsymbol{r}-\omega t)} = Ae^{i(\boldsymbol{k}\cdot(-\boldsymbol{r})-\omega(-t))} = \psi(-\boldsymbol{r},-t)$ which is exactly the wave function of a free particle that travels in the opposite direction in reversed time (denoted this imaginary free particle as Y). Therefore, the absolute value of X's wave function can be written as (it is also the probability density that X is at $\boldsymbol{r}$)

$$|\psi(\boldsymbol{r},t)|^2 = \psi^*(\boldsymbol{r},t)\cdot\psi(\boldsymbol{r},t) = \psi(-\boldsymbol{r},-t)\cdot\psi(\boldsymbol{r},t)$$

which is the multiplication of Y's wave function and X's wave function, which is non-zero only when both of them are non-zero. This strongly resembles the "intersection" in RA where X refers to the forward process and Y refers to the time-reversed backward process. And yet, free particles indeed travel in the quickest way.

Are these arguments above suggesting that the resonance-algorithm-like process is underlying these natural phenomena? Indeed, these thoughts are by no means scientifically solid, but it is deserved to be said if it offers any inspiration, reflection, or chance of discussions.

B. Extend Dijkstra's algorithm to give all possible shortest paths

The classic Dijkstra's algorithm only gives one shortest path, even if there are more equally-long shortest paths. Here we first illustrate this classic version and explain why it has this property; and then extend this classic version to give all possible shortest paths. Take the graph shown in Fig. 4 as an example, and look for shortest paths from node A to B. Initially, only the origin node A is in the set of "visited", denoted as set Ω. For each round, visited nodes' neighbors that are not

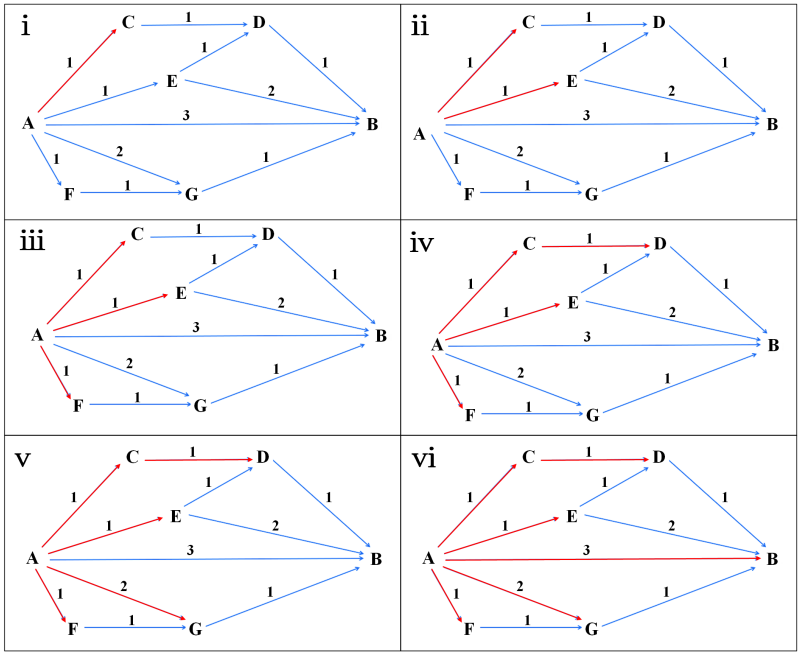

Figure 4. Illustration of the classic Dijkstra's algorithm.

included in Ω will be examined, and the one that has the shortest *tentative distance* will be added to Ω.

As shown in Fig. 4, in round i, node A's neighbors B, C, E, F and G are examined. We find that the distances from A to C, E and F are the smallest (namely, 1) but identical. Yet we should only put either C, E or F into Ω (usually the one with the smallest index, node C in this case). In round ii, A's and C's neighbors B, D, E, F and G are examined, and E enters Ω. In round iii, A's, C's and E's neighbors B, D, F and G are examined, and F enters Ω. Repeat, until all nodes are included in Ω; and then we have found the shortest path from the origin to any node. The reason why this algorithm gives only one shortest path is that the nodes in set Ω can only be accessed once. For example, after node D is included in Ω (accessed through node C), it cannot be accessed through node E. So, path A→E→D cannot be found, although it has the same length as A→C→D.

To solve this problem so that it gives all equally-long shortest paths, we extend the single parent node that is attached to each node, as in the classic algorithm, to a "parent nodes list" [32] (see codes at www.wuyichen.org/resalg). Furthermore, when a new node N enters Ω, we do not immediately go to the next round as before, but first examine each of N's neighbors that is the closest to N, denoted as node M (for convenience, denote M's current tentative distance as d, and the newly-calculated distance from the origin to M through N as d'): (1) if $d' < d$, M's tentative distance will be updated to d', and M's parent nodes list will be reset such that it only includes N; (2) if $d' = d$, N will be added to M's parent node list; (3) if $d' > d$, nothing happens.

For example, in round ii, when E enters Ω, we should then examine D (since D is closer to E among all of E's neighbors, namely D and B). Currently, C is D's only parent node. But now we find that the distance from the origin to D through E is equal to the current tentative distance of D (namely, the distance of the path A→C→D). So, we add node E into D's parent nodes list (which includes C and E now).

Continue this process until all nodes are included in Ω. At the end, all possible shortest paths can be obtained by recursively tracing the parent nodes list of each node.

References

1. Deo N, Pang CY.
   Shortest-path algorithms: Taxonomy and annotation.
   Networks. 1984;14:275–323.
2. Zhang X, Mahadevan S.
   A Bio-Inspired Approach to Traffic Network Equilibrium Assignment Problem.
   IEEE Transactions on Cybernetics. 2018 April;48(4):1304–1315.
3. Bast H, Delling D, Goldberg A, Müller-Hannemann M, Pajor T, Sanders P, et al.
   Algorithm Engineering.
   In: Route Planning in Transportation Networks. Springer, Cham; 2016. p. 19–80.
4. Delling D, Sanders P, Schultes D, Wagner D.
   Algorithmics of Large and Complex Networks.
   In: Engineering Route Planning Algorithms. Springer Berlin Heidelberg; 2009. p. 117–139.
5. Medhi D, Ramasamy K.
   In: Chapter 2 - Routing Algorithms: Shortest Path, Widest Path, and Spanning Tree. second edition ed. The Morgan Kaufmann Series in Networking. Boston: Morgan Kaufmann; 2018. p. 30–63.
6. Mahboubi H, Masoudimansour W, Aghdam AG, Sayrafian-Pour K.
   An Energy-Efficient Target-Tracking Strategy for Mobile Sensor Networks.
   IEEE Transactions on Cybernetics. 2017 Feb;47(2):511–523.
7. Dijkstra F, van der Ham J, Grosso P, de Laat C.
   A path finding implementation for multi-layer networks.
   Future Generation Computer Systems. 2009;25:142 – 146.
8. Sun X, Liu Y, Yao W, Qi N.
   Triple-stage path prediction algorithm for real-time mission planning of multi-UAV.
   Electronics Letters. 2015;51:1490–1492.
9. Xiao P, Ju H, Li Q, Xu H, Lu C.
   Task Planning of Space Maintenance Robot Using Modified Clustering Method.
   IEEE Access. 2020;8:45618–45626.
10. Dijkstra EW.
    A note on two problems in connexion with graphs.
    Numerische Mathematik. 1959;1:269–271.
11. Peng W, Hu X, Zhao F, Su J.
    A Fast Algorithm to Find All-Pairs Shortest Paths in Complex Networks.
    Procedia Computer Science. 2012;9:557 – 566.
12. Lu X, Camitz M.
    Finding the shortest paths by node combination.
    Applied Mathematics and Computation. 2011;217(13):6401–6408.
13. Orlin JB, Madduri K, Subramani K, Williamson M.
    A faster algorithm for the single source shortest path problem with few distinct positive lengths.
    Journal of Discrete Algorithms. 2010;8(2):189–198.
14. Bang-Jensen J, Gutin G.
    Digraphs: Theory, Algorithms and Applications.
    London: Springer-Verlag London; 2009.
15. Chan TM.
    All-Pairs Shortest Paths with Real Weights in $O(n^3 / \log n)$ Time.
    Algorithmica. 2008;50(2):236–243.
16. Liu X, Li J, Hu F, Yang C.
    Obstacle Induced Stochastic Tree for Fast Path Planning.
    In: 2019 IEEE International Conference on Real-time Computing and Robotics (RCAR); 2019. p. 499–503.
17. Noto M, Sato H.
    A method for the shortest path search by extended Dijkstra algorithm.
    In: 2000 IEEE international conference on systems, man and cybernetics.; 2000. p. 2316–2320.
18. Sastry VN, Janakiraman TN, Mohideen SI.
    New Algorithms For Multi Objective Shortest Path Problem.
    OPSEARCH. 2003;40:278–298.

19. Storandt S.
Quick and Energy-Efficient Routes: Computing Constrained Shortest Paths for Electric Vehicles.
In: Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science. New York, NY, USA: Association for Computing Machinery; 2012. p. 20–25.
20. Zhang S, Liu X.
A new algorithm for finding the k shortest transport paths in dynamic stochastic networks.
Journal of Vibroengineering. 2013;15:726–735.
21. Zhu D, Huang H, Yang SX.
Dynamic Task Assignment and Path Planning of Multi-AUV System Based on an Improved Self-Organizing Map and Velocity Synthesis Method in Three-Dimensional Underwater Workspace.
IEEE Transactions on Cybernetics. 2013 April;43(2):504–514.
22. Moon S, Oh E, Shim DH.
An Integral Framework of Task Assignment and Path Planning for Multiple Unmanned Aerial Vehicles in Dynamic Environments.
Journal of Intelligent & Robotic Systems. 2013;70(1):303–313.
23. Pascoal MMB, Resende M.
The minmax regret robust shortest path problem in a finite multi-scenario model.
Applied Mathematics and Computation. 2014;241:88–111.
24. Ohtsubo Y.
Stochastic shortest path problems with associative accumulative criteria.
Applied Mathematics and Computation. 2008;198(1):198–208.
25. Galán-García JL, Aguilera-Venegas G, Galán-García MÁ, Rodríguez-Cielos P.
A new Probabilistic Extension of Dijkstra's Algorithm to simulate more realistic traffic flow in a smart city.
Applied Mathematics and Computation. 2015;267:780 – 789.
26. Xu MH, Liu YQ, Huang QL, Zhang YX, Luan GF.
An improved Dijkstra's shortest path algorithm for sparse network.
Applied Mathematics and Computation. 2007;185(1):247–254.
27. Hershberger J, Maxel M, Suri S.
Finding the k Shortest Simple Paths: A New Algorithm and Its Implementation.
ACM Trans Algorithms. 2007;3:45.
28. Hemalatha S, Valsalal P.
Identification of Optimal Path in Power System Network Using Bellman Ford Algorithm.
Modelling and Simulation in Engineering. 2012;2012:913485.
29. Waterman MS.
Sequence alignments in the neighborhood of the optimum with general application to dynamic programming.
Proceedings of the National Academy of Sciences. 1983;80:3123–3124.
30. Jiang M, Chen Y, Zhang Y, Chen L, Zhang N, Huang T, et al.
Identification of hepatocellular carcinoma related genes with k-th shortest paths in a protein–protein interaction network.
Mol BioSyst. 2013;9:2720–2728.
31. Eppstein D.
Finding the k Shortest Paths.
SIAM Journal on Computing. 1998;28:652–673.
32. Wang S, Li A.
Multi-adjacent-vertexes and Multi-shortest-paths Problem of Dijkstra Algorithm.
Computer Science. 2014;41:217–224.
33. Mohanta K.
Comprehensive Study on Computational Methods for K-Shortest Paths Problem.
International Journal of Computer Applications. 2012;40:22–26.
34. Ziggelaar A.
The Sine Law of Refraction Derived from the Principle of Fermat—Prior to Fermat? The Theses of Wilhelm Boelmans S. J. in 1634.
Centaurus. 1980;24:246–262.

35. Schlosshauer M.
Decoherence, the measurement problem, and interpretations of quantum mechanics.
Rev Mod Phys. 2005 2;76:1267–1305.
36. Helgason RV, Kennington JL, Stewart BD.
The one-to-one shortest-path problem: An empirical analysis with the two-tree Dijkstra algorithm.
Computational Optimization and Applications. 1993;2:47–75.